Supporting Programming by Analogy in the Learning of Functional Programming Languages

Supporting Programming by Analogy in the Learning of Functional Programming Languages

Jon Whittle Alan Bundy
Dept. of Artificial Intelligence, University of Edinburgh
80 South Bridge, Edinburgh EH1 1HN, UK.
+44 (0)131 650 2725

Helen Lowe*
Dept. of Computer Studies, Napier University
Craiglockhart, 219 Colinton Road, Edinburgh EH14 1DJ
+44 (0)131 455 4215
Email: jonathw@dai.ed.ac.uk

Keywords: Programming Language Learning, Learning Environments, Analogy

April 2, 1997

Abstract

This paper examines the learning of the functional programming language Standard ML. A common technique used by novices is programming by analogy whereby students refer to similar programs that they have written before or have seen in the course literature and use these programs as a basis to write a new program. We present a novel editor for ML which supports programming by analogy by providing a collection of editing commands that transform old programs into new ones. Each command makes changes to an isolated part of the program. These changes are propagated to the rest of the program using analogical techniques. Many commands are at a level high enough to provide guidance to the novice during program development. We observed a group of novice ML students to determine the most common programming errors in learning ML and restrict our editor such that it is impossible to commit these errors. In this way, students encounter fewer bugs and so their rate of learning increases.

*The first author was supported by an EPSRC studentship. The second and third authors are supported by EPSRC grant GL/L/11724

1 Introduction

Functional programming languages such as LISP, ML and Hope are increasingly being used in academe and industry. Many universities now teach functional languages as a key part of their software engineering programme. However, the teaching of such languages presents problems. Functional languages involve abstract concepts such as recursion which are difficult to learn ([APF88]). Many experiments have been carried out that suggest that students overcome these difficulties by using analogy in the early stages of programming [PA85, WB95]. Given a program to write, novices refer to similar programs they have written before or seen in the course literature. They then use the old program as a basis to construct the new one. We have conducted our own informal experiment with a group of 30 novice ML students which involved observations of the students over the course of a semester and in-depth interviews with two of the students. This provided additional evidence of programming by analogy [Whi96].

We have implemented a program editor, *CYNTHIA*, for Standard ML that supports programming by analogy. ML is a typed, functional language incorporating extensive use of pattern matching and recursion. The user edits a program by applying a sequence of editing commands. These commands allow the user to make isolated changes to an existing (partial) program. *CYNTHIA* then propagates these changes automatically to the rest of the program hence producing a new one. It does this using an analogical mechanism. In addition, each editing command guarantees certain aspects of correctness so that any program produced is free of certain kinds of bugs. As a simple example of the idea, suppose the user is writing a function, *count*, to count the number of nodes in a tree. He has seen a program before, *length*, to count the number of items in a list and uses *length* as a starting point:[1]

```
fun length nil = 0
|   length (x::xs) = 1 + (length xs);
```

We show how *length* could be edited into *count*.

1. The user may indicate any occurrence of *length* and invoke the RENAME command to change *length* to *count*. *CYNTHIA* then uses its analogical mechanism to change all other occurrences of *length* to *count*:
   ```
   fun count nil = 0
   |   count (x::xs) = 1 + (count xs);
   ```
2. We want to count nodes in a tree so we need to change the type of the parameter. Suppose the user indicates *nil* and invokes CHANGE TYPE to change the type to *tree*. *CYNTHIA* propagates this change by changing *nil* to (*leaf n*) and changing :: to *node*:
   ```
   fun count (leaf n) = 0
   |   count (node(xs,ys)) = 1 + (count xs);
   ```
   Note that the program no longer contains x. Instead, a new variable *ys* of type *tree* has been introduced. In addition, (*count ys*) is made available for use as a recursive call in the program.
3. It remains to alter the results for each pattern. 0 is easily changed to 1 using CHANGE RESULT. If the user then clicks on 1 in the second line, a list of terms appear which include (*count ys*). Selecting this term produces the final program:
   ```
   fun count (leaf n) = 1
   |   count (node(xs,ys)) = (count ys) + (count xs);
   ```

Under this methodology, all programs are constructed using editing commands that induce incremental changes. There are both low and high-level commands. Low-level commands make only very simple changes to the program – e.g. CHANGE RESULT in 3. High-level commands affect the overall structure of the program – e.g. changing the type in 2. See §3.1 for details. The ideal way to use the editor is to apply the highest level commands first and then use lower-level commands to fill in the details. This encourages the user to think about his programs in high-level terms. This concept of editing can be used in a number of ways. As well as supporting programming by analogy, mistakes discovered during program development or program testing can be rectified easily.

We aim our system primarily at novices. However, *CYNTHIA* is general enough to allow complex, practical programs to be produced. It is unlike many tutoring systems (e.g. [BGM94]) that are restricted to a small number of toy examples. This means the novice has the freedom to experiment and enables continued support once the novice has become more expert.

[1] *::* is the ML list operator cons

2 The Design of $CYNTHIA$

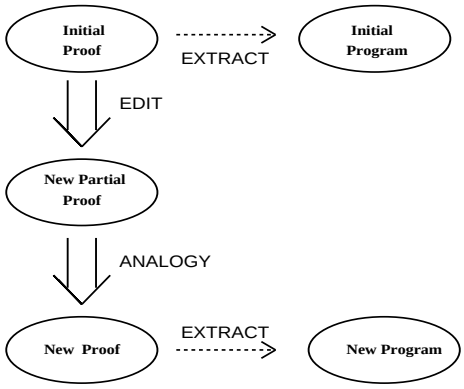

Figure 1: Editing Programs in $CYNTHIA$

We wish $CYNTHIA$ programs to be guaranteed correct in some respects. It is natural, therefore, to base the design around established techniques from logic and proof theory which is a flexible and powerful way of reasoning about the correctness of programs. [How80] identifies the necessary machinery to set up a one-to-one correspondence between functional programs and mathematical proofs in a constructive logic. This isomorphism has been used as the basis for program verification and synthesis. For instance, within the paradigm of program verification, given a program, we can prove it correct by proving the corresponding theorem in the constructive logic. As an example, given a program to append two integer lists together, we could formulate a theorem[2]

$$\forall x : list(int)\ \forall y : list(int)\ \exists z : list(int)\ (\forall e : int\ e \in z \leftrightarrow e \in x \wedge e \in y) \tag{1}$$

This theorem or *specification* states the existence of a list z that contains all elements of x and y and no others. Hence, a possible z is x @ y[3]. Suppose we have a proof of this specification in a constructive logic. We can extract a functional program from this proof such that the program is guaranteed to be correct with respect to the specification – i.e. it will compute x @ y. This is called the *proofs-as-programs* paradigm. It enables us to construct programs that are correct in some way.

We use a restricted form of this idea where the specification does not describe the full behaviour of the corresponding program but instead states the number and the type of input arguments and the type of the output argument. For example, *append* would have a spec $list(int) \rightarrow list(int) \rightarrow list(int)$. Every program is associated with a corresponding specification and *synthesis proof*. Our proofs are written in the proof editor *Oyster* [HS90] which is based on a constructive logic known as Martin-Löf's Type Theory [ML79]. The synthesis proof essentially guarantees the correctness of the program extracted from it. The more detailed the specification, the more we can guarantee about the program. Our simple specifications prove that $CYNTHIA$ programs are syntactically correct, well-typed, well-defined and terminating – see §3.2 for more details.

The design of $CYNTHIA$ is depicted in Figure 1. Note that editing commands directly affect the synthesis proof and only affect the program indirectly. The user begins with an initial program and a corresponding synthesis proof. These may be incomplete. Editing commands make changes to an isolated part of the synthesis proof. This yields a new partial proof which may contain gaps or inconsistencies. To fill in these gaps and resolve inconsistencies, we use an analogical mechanism. This mechanism *replays* the proof steps in the original (source) proof to produce a new (target) proof. During this replay, the changes induced by the editing command are propagated throughout the proof. Once gaps in the target proof have been bridged, a new program is extracted. This program incorporates the user's edits is guaranteed correct.

Refer to the example in §1. We briefly explain how the analogy works in 2. The user has selected *nil* and indicated a change of type. This will change *nil* to the counterpart for trees, $(leaf\ n)$. We are now

[2] $X : T$ means X is of type T

[3] In fact, *append* is just one program that would satisfy the specification. We can write the specification to any level of detail.

at the 'New Partial Proof' stage in Figure 1. The proof will be incomplete and to bridge this gap, :: will need to be changed to *node*. The analogy replays the original proof and changes proof steps along the way so that $x :: xs$ becomes $(node(xs,\ ys))$ and the recursive call $(count\ ys)$ is made available for future use.

In general, constructing proofs by analogy is a difficult task [MW96]. Because we are restricted to specifications involving a limited amount of detail, the proofs are simpler and so the analogy becomes a viable option in a practical, real-time system such as ours.

$CYNTHIA$ is equipped with an interface that hides the proof details from the user. As far as the user is aware, he is editing the program directly. In this way, the user requires no knowledge of logic and proof.

3 Increasing the Learning Rate

Over a period of three months we conducted observations of a group of 30 novice ML students from Napier University to ascertain what problems presented themselves when learning ML. The students were observed writing programs during weekly tutorial sessions. In addition to informal observations, their interactions with ML were scripted and analysed. The students completed questionnaires relating their experiences and at the end of the course, two students were interviewed in depth. We describe here the main drawbacks to learning ML and how $CYNTHIA$ addresses them. The students used version 0.93 of New Jersey ML and so our comments refer to this version.

3.1 Program Transformation

To provide the maximum support for programming by analogy, the editing commands in $CYNTHIA$ are structured into low-level commands for making very small changes and high-level commands for changing the overall program structure. Not only does this approach constitute a powerful way of transforming programs but it also encourages the novice to follow a top-down approach to programming – deciding on the high-level structure first and then filling in the details.

3.1.1 Low-Level Commands

These are commands that do not affect the datatype of the function being defined. Nor do they affect the recursion the function is defined by. This means that the analogy needed to produce a new program is fairly straightforward. The low-level commands are invoked to change the arity of a function, reorder arguments, rename terms, add or remove conditional expressions, add local variable declarations or simply replace terms. In addition, each SML construct has a corresponding command for introducing or removing the construct. Figure 2 gives an idea of some low-level commands used to transform, *rev* a function for reversing lists, into *delete* for deleting an element from a list. The commands are in upper case. @ is the ML append operator. The first step renames the function and adds an extra argument. ADD ARGUMENT is invoked by indicating an occurrence of *rev* and then analogy gives all other occurrences of *rev* an additional argument too. ADD IF..THEN..ELSE places a case-split at the designated position, duplicating whatever is below the current position in the original program. CHANGE RESULT is used to edit the result for one of the patterns – e.g. to remove @ in $(delete\ xs\ e)$ @ $[x]$ giving $(delete\ xs\ e)$.

The high-level commands consist of those for changing the recursion and those for changing the type of an argument.

3.1.2 Definition by Patterns

By *definition by patterns* we mean the common practice as used in ML whereby functions are defined by pattern matching (see *rev* for example in §3.1.1). We observed that novices often have difficulty in deciding upon the correct definition by patterns for a function. They are capable in simple cases where the function has only one argument that is pattern matched against, but become lost when more than one argument is pattern matched or when the pattern used is non-standard. A simple function that pattern matches multiple arguments would be an *nth* function to return the nth element in a list.

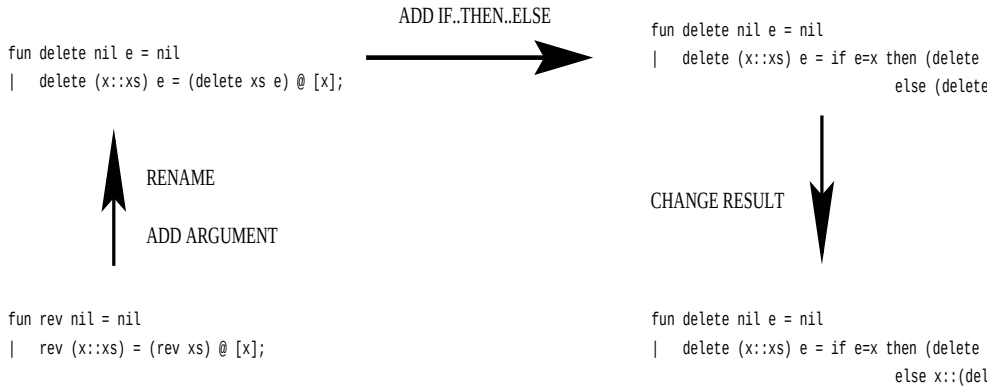

Figure 2: Low-Level Commands

We have commands CREATE PATTERN and REMOVE PATTERN which allow the user to build up non-standard patterns by combining a number of standard ones. We have implemented a version of a technique used in ALF [Coq92]. The user can highlight an object of a certain datatype. The application of the CREATE PATTERN command splits the object into a number of patterns - one for each constructor function used to define the datatype. Hence, CREATE PATTERN on *x:list* below

```
fun f(x,l)=..
```

produces two patterns:

```
fun f(nil,l)=..
|   f(h::t,l)=..
```

Non standard patterns can be defined by applying the command a number of times. Highlighting t and applying CREATE PATTERN gives:

```
fun f(nil,l)=..
|   f(h::nil,l)=..
|   f(h::h2::t1,l)=..
```

This can be done for any datatype by using the definition of the type as encoded in ML. Suppose l is of type *tree* then we can split l in the second pattern to give:

```
fun f(nil,l)=..
|   f(h::nil,(leaf x))=..
|   f(h::nil,(node(l1,l2)))=..
|   f(h::h2::t1,l)=..
```

We do not use the same underlying theory as ALF but use the constructive logic already available in our proof system. The result is the same, however. We have additionally implemented a REMOVE PATTERN command for removing patterns (ALF does not have this). This command can also be applied to patterns introduced by the ML construct `case`[4].

3.1.3 Recursion

Recursion is well-known to be a difficult concept to learn. Novices can have considerable difficulty with even primitive recursion schemes. However, an introductory course will also introduce non-standard schemes involving accumulators, multiple recursion, course-of-values recursion and nested recursion. To help novices to learn non-standard recursions, the commands ADD RECURSIVE CALL and REMOVE RECURSIVE CALL encourage them to think about which recursive calls are needed for the task at hand. *CYNTHIA* maintains a list of recursive calls that are currently available to the user. When the user is required to enter a term, these recursive calls are among the options presented to the user. He can pick

[4] `case` splits a term into patterns within a definition allowing nested patterns. It does not allow recursive calls

one using the mouse without any need for further typing. The user can change the list of recursive calls by the commands mentioned above. The idea is that the user first decides upon what kind of recursion he should use. He can then use these commands to set up the basic structure within which to use them. Other commands can be used to fill in the details.

As an example of how the commands can be used, consider trying to produce the function *msort*:

```
fun msort nil = nil
|   msort (x::nil) = x
|   msort (x::h::t) = merge (msort (evenl (x::h::t)))
                            (msort (x :: (evenl (h::t))));
```

evenl returns the elements in a list at even positions. *merge* joins two lists by repeatedly taking the smaller of the two heads. The ideal way to proceed is to decide upon the program structure to begin with by forming a definition by patterns and then introducing the recursive calls necessary. We can define a function *msort* with one argument (of type integer list). Figure 3 shows the edits then needed to produce the recursive structure. CREATE PATTERNs are applied to provide the required patterns in the definition. Then the available recursive calls are changed. The remaining details are filled in by low-level commands. To avoid restricting the user, he is not forced to produce programs in this top-down fashion. The result is independent of the order of execution of commands.

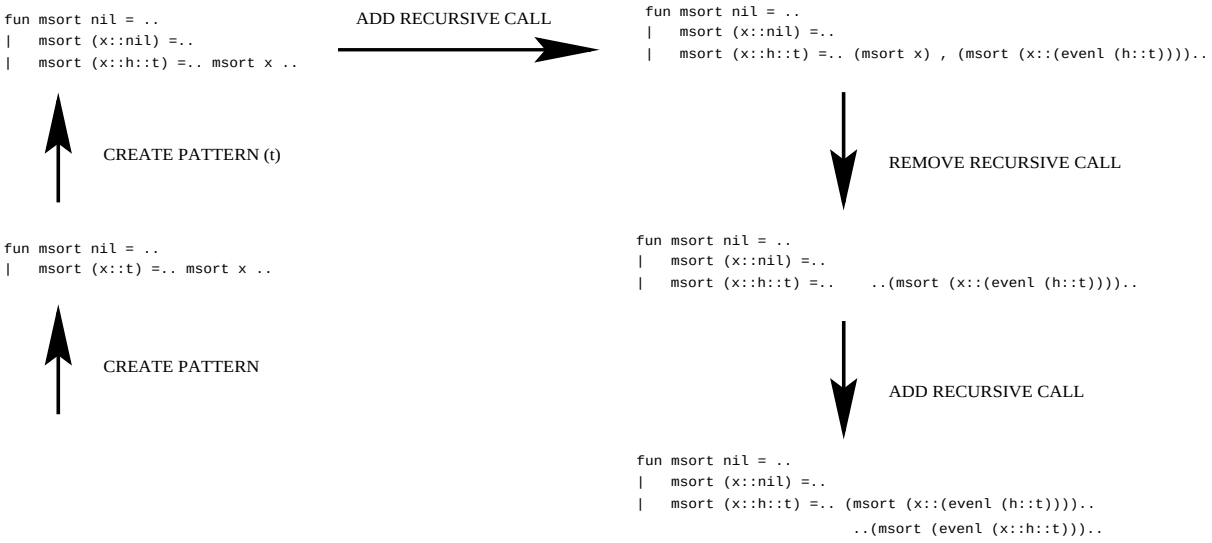

Figure 3: Writing *msort*

3.1.4 Changing Type

A major drawback of current learning processes is that novices can ignore datatypes. ML is equipped with a type inference engine which automatically derives (if possible) the type of the top-level function. Although advantageous in that users need not explicitly state the type of each term, novices can ignore types and be unaware of type inconsistencies which may arise. This results in unhelpful error messages from the compiler and confusion. For instance, in the function:

```
fun length nil = nil
|   length (x::xs) = 1 + (length xs);
```

we have an example of a simple but commonly-made error whereby the output type in the first line is list but is an integer in the second. It can often be difficult to pinpoint exactly where the error has occurred.

For this reason, we insist that the user declares the type of a function before anything else. This forces the novice to think about the types hence reducing type errors when writing the rest of the program.

Once the top-level type has been given, the types of terms in other parts of the program are determined and hence need not be given by the user.

During the course of a program the user may realise he has stated the top-level type incorrectly. Or he may want to change the top-level type of an old program to produce a new one. $CYNTHIA$ provides quite advanced facilities for doing this. Changing the output type of a function or changing the type of a non-recursive argument does not present too many problems. It can result in inconsistencies in the proof (ill-typed terms) but this can be dealt with – see §3.3. The real challenge comes when changing the type of an argument that is pattern matched against since the pattern needs to be changed. If we wanted to change the type of the first argument in *length* from list to tree, we can invoke CHANGE TYPE to edit *length* into

```
fun length (leaf n) = nil
|   length (node(xs,ys)) = 1 + length xs;
```

In this case, $CYNTHIA$ will also add the recursive call *length ys* to the list available. This could then be used by the user. More complicated examples arise when CREATE PATTERN has been applied more than once. If our original program had been:

```
fun app2 nil l2 = l2
|   app2 (x::nil) l2 = (x::l2)
|   app2 (x1::x2::xs) l2 = x1 :: (app2 xs l2);
```

it is not clear what the new pattern should be after a change of type of the first argument to *tree*. $CYNTHIA$ looks for a mapping between the old and new datatype definitions and uses heuristics to select a mapping if necessary. This mapping is then applied to the old pattern definition to produce a new one. The details are too complicated to go into here. The upshot is that the user can highlight any argument in the function and enter a new type for that argument. $CYNTHIA$ will then propagate the effect of this type change to the rest of the program, changing the pattern by which the function is defined if necessary.

3.2 Reducing the Number of Programming Errors

One of the main purposes of our experiment was to identify the kinds of programming errors that novice ML users encounter. Our results suggest that the learning rate is severly affected by these errors. The most common errors were syntax errors and type errors. $CYNTHIA$ disallows such errors in its programs. The students found it particularly difficult to pinpoint the source of a type error. Although ML does type checking at compile time and spots type inconsistencies, the system messages provide little or no help in rectifying the problem. $CYNTHIA$ also incorporates a type checker. The ML type checker is not invoked until compile time. In $CYNTHIA$, however, the type checker is called as each new term is entered. Hence, the user receives immediate feedback on whether a term is well-typed. In addition, given the type of the top-level function that the user has already supplied, $CYNTHIA$ can tell the user what the type of a term should be *before* he enters it. In this way, the number of type errors is reduced considerably. All programs in $CYNTHIA$ are guaranteed to be well-typed.

A major source of errors in recursive programs is non-termination [BGB91]. An example of a non-terminating function is

```
fun gcd x y = if x=y then x
              else gcd (x-y) y;
```

Note the result of calling $gcd(2,3)$: $gcd(2,3) = gcd(-1,3) = gcd(-4,3) = ...$ Termination errors are not spotted at compile time but result in run-time errors. Although less frequent than type errors they are usually more catastrophic. In $CYNTHIA$, the user is restricted to terminating programs[5]. Because termination checking is undecidable it is impossible to restrict $CYNTHIA$ such that one can define *all* terminating programs and no others. There will always be terminating functions that cannot be produced with $CYNTHIA$. For guaranteeing termination of a wide range of functions our basic design is augmented by a termination checker based on Walther Recursion [MA96]. Walther recursive functions

[5] Occasionally, non-terminating programs can be useful. One could envisage, however, a facility for overriding termination restrictions in these small number of cases.

form a decidable subset of all terminating functions. Checking if a function is Walther recursive is easily computable and hence ideal for a real-time system. The set of Walther recursive functions is also wide enough to be of real practical use – including most commonly occurring examples of course-of-values, nested and multiple recursions. The functions that can be produced in $CYNTHIA$ form precisely the set of Walther recursive functions.

As mentioned earlier, a common way of defining ML programs is by pattern matching. A source of errors in our analysis was that students wrote programs that were not well-defined – i.e. the pattern for an argument did not exhaustively cover the domain of the datatype with no redundant matches. ML spots such errors at compile time displaying a warning message. Although ML does not consider ill-definedness as an error, it is commonly believed that it is good programming practice to write well-defined programs. Otherwise, there can be serious run-time errors. Students were found to ignore the warnings given by ML because they are not explicitly flagged as errors. We feel, however, that students would make less errors if their programs were all well-defined. Hence, in $CYNTHIA$ the editing commands guarantee well-definedness.

In addition to these guarantees, any program defined in $CYNTHIA$ is syntactically correct.

3.3 Supporting the Detection of Errors

Although the aim is, in general, to produce a valid program at each stage of editing, this is not always possible. Some editing commands will invalidate parts of the program. There are two main ways this can happen. The first arises when removing part of a program. If a section, S of a program is deleted and a subsequent part of the program, P depends on S then P can no longer be valid. For instance, suppose a function has a number of arguments including x and that somewhere in the program the user has made a casesplit on x, say `if x=0 then ...else ...`. If he removes x from the input list, a program that applies the same casesplit would not be a valid ML program because x no longer exists. We overcome this by allowing *partial programs*. A partial program is a program that may contain variables that are not referenced elsewhere. They will always appear highlighted to the user. This highlighting tells the user that he must change the variable before the program is valid. This highlighting retains as much as possible of the program so that frustrating retyping is not necessary whilst additionally telling the user exactly which parts of the program must be changed next. It provides an invaluable way of pinpointing exactly which parts of the program will be affected by edits.

The other situation where this is used is when the type of an argument is changed. In ML, it is notoriously difficult to pinpoint the source of a type error. In $CYNTHIA$, however, every term that is entered is type-checked at entry time. If the term is not of the required type, the user will be told and will not be allowed to enter the term. More generally, if an application of the CHANGE TYPE command makes some part of the program ill-typed, $CYNTHIA$ will not only highlight the offending term to alert the user but will also tell the user what the type should be so that when he changes the highlighted term he is told what type the new term should belong to.

Note that the proofs-as-programs paradigm is a natural way in which to implement this mechanism. No extra checks are needed to highlight terms. Highlighted parts of the program just correspond to proof rules that have failed to apply. Similarly, highlighting of ill-typedness means that a proof obligation to prove the well-typedness has failed to be proved and the user is alerted.

4 Related Work

The work closest to our own is the *recursion editor* presented in [BGB91]. In fact, this was one of the original inspirations. The recursion editor is an editor for writing terminating programs. Like our system, edits are made by invoking editing commands. $CYNTHIA$'s commands are more general than those in the recursion editor. In the recursion editor, only a very restricted subset of recursive programs could be produced. The recursion editor is very sensitive to the order in which commands are performed. If they are performed in the wrong order, it may be difficult or impossible to recover. Our proofs-as-programs design overcomes this by allowing greater flexibility because it keeps track of the dependencies within the same program and between different programs. Our proof design also allows us to locate errors in the program easily. [BGB91] makes no consideration of datatypes.

Some work has been done on programming using schemata [KLS89, GH92]. This is similar in spirit to our low and high-level commands as the user follows a top-down approach. However, previous attempts are limited to a small range of programs. Our editor is much more general providing a range large enough to be of real practical use. The techniques editor TED [BG96] features a program transformation perspective but it has no strong theoretical foundations and is therefore much less powerful than $CYNTHIA$.

We do not consider the problem of retrieving a previous example – see [Web96] which indicates that students tend to solve problems in analogy to the most recent problem they have attempted even though this may not be the best starting point. Although we do not address this issue, our system is at least general enough such that a poor choice of base problem should not prevent a correct, albeit sub-optimal, transformation sequence leading to a solution. As yet, the use of metacognitive tools forcing students to think about their problem solving process has not been very effective [Web96]. Although $CYNTHIA$ encourages students to think about such issues, they retain control to explore unconventional paths.

There exist many editors that guarantee syntactic correctness (e.g. [KW87]). We are aware of no editor that provides the guarantees that we do.

5 Conclusions and Further Work

This paper has presented an editor for producing correct functional programs. It builds upon ideas in [BGB91]. The editor is intended to be a suitable vehicle for novices to learn the language ML. Its high-level commands provide guidance to the user and the user is prevented from making certain kinds of programming error.

Our work can be seen on a number of levels. First, as an educational aid, it provides support for novices learning a language by reducing the effort needed to produce correct programs but without restricting the user to text book solutions. Second, as a support tool for ML, it is a way to quickly edit existing programs without introducing unnecessary bugs. Third, it is an interesting application of ideas from the field of automated reasoning.

In the near future, we intend to develop a GUI for $CYNTHIA$ and explore further ways in which guidance can be given to the user. We also hope to explore ways in which stronger guarantees of correctness such as behavioural correctness can be incorporated. The system is due to be tested on ML students from September 1997 onwards.

Acknowledgements: We are grateful to Andrew Cumming for help and discussions on the experiment with his ML students. We also thank Paul Brna for insightful comments on this paper.

References

[APF88] J. R. Anderson, P. Pirolli, and R. Farrel. Learning to program recursive functions. *The Nature of Expertise*, pages 153–183, 1988.

[BG96] P. Brna and J. Good. Searching for examples: An evaluation of an intermediate description language for a techniques editor. In P. Vanneste, K. Bertels, B. de Decker, and J-M. Jaques, editors, *Proceedings of the 8th Annual Workshop of the Psychology of Programming Interest Group*, pages 139–152. 1996.

[BGB91] A. Bundy, G. Grosse, and P. Brna. A recursive techniques editor for Prolog. *Instructional Science*, 20:135–172, 1991.

[BGM94] S. Bhuiyan, J. Greer, and G. I. McCalla. Supporting the learning of recursive problem solving. *Interactive Learning Environments*, 4(2):115–139, 1994.

[Coq92] Th. Coquand. Pattern matching with dependent types. In *Proceedings from the logical framework workshop at Bästad*, June 1992.

[GH92] T.S. Gegg-Harrison. Adapting instruction to the student's capabilities. *Journal of AI in Education*, 3:169–181, 1992.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[HS90] C. Horn and A. Smaill. Theorem proving and program synthesis with Oyster. In *Proceedings of the IMA Unified Computation Laboratory*, Stirling, 1990.

[KLS89] M. Kirschenbaum, A. Lakhotia, and L.S. Sterling. Skeletons and techniques for Prolog programming. Technical Report Tr-89-170, Case Western Reserve University, 1989.

[KW87] A. Kohne and G. Weber. Struedi: A lisp-structure editor for novice programmers. In *Human-Computer Interaction (INTERACT 87)*, pages 125–129, 1987.

[MA96] David McAllester and Kostas Arkoudas. Walther recursion. In *CADE-13*, pages 643–657. Springer Verlag, July 1996.

[ML79] Per Martin-Löf. Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Hanover, August 1979. Published by North Holland, Amsterdam. 1982.

[MW96] E. Melis and J. Whittle. Internal analogy in theorem proving. In M. A. McRobbie and J. K. Slaney, editors, *13th International Conference on Automated Deduction (CADE-13)*, pages 92–105. Springer, 1996. Also available from Edinburgh as DAI Research Paper No 803.

[PA85] P. L. Pirolli and J. R. Anderson. The role of learning from examples in the acquisition of recursive programming. *Canadian Journal of Psychology*, 39:240–272, 1985.

[WB95] G. Weber and A. Bögelsack. *Representation of Programming Episodes in the ELM model.* Ablex Publishing Corporation, Norwood, NJ, 1995.

[Web96] G. Weber. Individual selection of examples in an intelligent learning environment. *Journal of AI in Education*, 7(1):3–31, 1996.

[Whi96] J. N. D. Whittle. An analysis of errors encountered by novice ML programmers. Tech.rep, University of Edinburgh, 1996. In http: //www.dai.ed.ac.uk/ daidb/ students/ jonathw/ publications.html.